

resolute:

A Semi-Automated Finite Element Mesh Generation Routine

J.D. Chaffey and D.A. Greenberg

Ocean Sciences Division
Maritimes Region
Fisheries and Oceans Canada

Bedford Institute of Oceanography
P.O. Box 1006
Dartmouth, Nova Scotia
Canada B2Y 4A2

2003

Canadian Technical Report of Hydrography and Ocean Sciences 225



Fisheries and Oceans
Canada

Pêches et Océans
Canada

Canada

Canadian Technical Report of
Hydrography and Ocean Sciences 225

2003

resolute:
A Semi-Automated Finite Element Mesh Generation Routine

by

J.D. Chaffey and D.A. Greenberg

Ocean Sciences Division
Maritimes Region
Department of Fisheries and Oceans
Bedford Institute of Oceanography
P.O. Box 1006
Dartmouth, N.S.
B2Y 4A2

©Her Majesty the Queen in Right of Canada 2003
Cat. No. Fs 97-18/225E ISSN 0711-6764

Correct citation for this publication:

Chaffey, J.D. and D.A. Greenberg. 2003. resolute: A Semi-Automated Finite Element Mesh Generation Routine. Can. Tech. Rep. Hydrogr. Ocean Sci. 225: vi + 33 p.

Contents

Abstract / Résumé	vi
1 Introduction	1
2 Installation	2
2.1 What you need	2
2.2 The Makefile	3
2.3 The PROJ library	3
2.4 Compilation	4
2.5 Installation	4
3 Usage	5
3.1 Before running	5
3.1.1 Filenames	5
3.1.2 Parameters	5
3.2 Running the script	7
4 Component Programs	9
4.1 mesh_init	9
4.2 nei_2ll	9
4.3 nei_2xy	10
4.4 nodtopoly	11
4.5 redepc	11
4.6 redep_triangle	11
4.7 regular_grid	12
4.8 reshape_node	12
4.9 set_triangle_area	13
4.10 showme	14
4.11 triangle	14
4.12 trngl2nei	15
5 File formats	16

5.1	.area files (triangle)	16
5.2	.bat files (triangle, NML)	16
5.3	.ele files (triangle)	16
5.4	.ele files (NML)	17
5.5	.nei files (Trigrd)	17
5.6	.node files (triangle)	17
5.7	.nod files (NML)	17
5.8	.nod files (Trigrd)	18
5.9	.poly files (triangle)	18
A	The MAKE_TRIANGLE_MESH Script	20
B	Examples of the script in action	26
C	Web Sites	33
C.1	resolute	33
C.2	PROJ	33
C.3	Triangle	33
C.4	TriGrid	33

List of Figures

1	The triangulated depth mesh.	27
2	The mesh after triangulation of the regular grid.	28
3	The mesh after the first refinement.	29
4	The mesh after the second refinement.	30
5	The mesh after the third refinement.	31
6	The mesh after the third refinement when allowing the use of Steiner points on the boundary.	32

Abstract

Chaffey, J.D. and D.A. Greenberg. 2003. *resolute*: A Semi-Automated Finite Element Mesh Generation Routine. Can. Tech. Rep. Hydrogr. Ocean Sci. 225: vi + 33 p.

The software package *resolute* generates a two dimensional finite element mesh for use in ocean modelling. This report describes a set of scripts and programs that creates a finite element mesh from a list of positions that gives the mesh boundaries and a list of random positions with depth data.

Résumé

Chaffey, J.D. and D.A. Greenberg. 2003. *resolute*: A Semi-Automated Finite Element Mesh Generation Routine. Can. Tech. Rep. Hydrogr. Ocean Sci. 225: vi + 33 p.

Le progiciel *resolute* produit un maillage d'éléments finis bidimensionnel utilisé pour modéliser l'océan. Dans ce rapport nous présentons une collection de scripts et de programmes permettant de créer un maillage d'éléments finis à partir d'une liste de positions qui définissent les limites du maillage et d'une liste de positions aléatoires auxquelles sont associées des profondeurs.

1 Introduction

The `make_triangle_mesh` script is a Perl program that generates a 2-D finite element mesh for use in ocean modelling using the `triangle` mesh generator and several other programs to create a mesh that satisfies a variety of requirements. The purpose of this report is to provide details of the usage and execution of this set of routines. However, the details of the triangulation methods and characteristics are given minimal coverage and the user should refer to that program's own web site (see Section 2.1) for a more complete description.

This report is divided into these main sections:

- directions on obtaining and installing the necessary programs;
- a description of the step-by-step process of creating and refining a finite element mesh;
- a description of the purpose and usage of the programs used by `make_triangle_mesh`;
- a description of the data file formats; and
- an step-by-step example of `make_triangle_mesh` producing a finite element mesh.

2 Installation

2.1 What you need

Of course, the first thing that is needed is the mesh generation package. It is available for download by ftp from `starfish.mar.dfo-mpo.gc.ca` and the package to download is `/pub/ocean/dgreenbe/resolute_1.1.tar.gz`. The package is comprised of a Perl script plus a number of programs. Most of the programs were written by the authors and perform the tasks of converting file formats, interpolating data values and calculating parameters for the triangulation. These programs are:

<code>mesh_init</code>	<code>nei_211</code>	<code>nodtopoly</code>
<code>redep_triangle</code>	<code>regular_grid</code>	<code>reshape_node</code>
<code>set_triangle_area</code>	<code>trngl2nei</code>	

(Descriptions of these programs can be found in Section 4.)

A C compiler is needed to compile the programs. The GNU C compiler, `gcc`, has been used and is included with all the major Linux distributions. It is also available from <http://gcc.gnu.org/>.

The `make_triangle_mesh` script was written with version 5 of Perl and tested on several different versions of RedHat and Mandrake linux. Most, if not all, distributions of Linux and Unix have Perl, but if you want to upgrade to the latest version, you can download it from <http://www.perl.com>.

In addition, two freely available software packages are required, both of which are included in the distribution. Only brief descriptions of these programs will be given in this document as they both have documentation available either online or included in their sub-directories.

The first is `triangle`, the program that generates the mesh through Delaunay triangulation. It is available over the web, from <http://www.cs.cmu.edu/~quake/triangle.html>.

The technical details of `triangle` will not be covered here, but references describing its implementation and some help files can be found on its website. Included in the `triangle` package is the `showme` program which can be used to view the results from `triangle`.

Also required is a library of cartographic projection software, `PROJ`. This library is used by the programs `mesh_init`, `nei_211` and `nei_2xy`, which convert position data from `[longitude latitude]` to `[x y]`. However, if the data you are using is not in `[longitude latitude]` format, then these two conversion programs are not run and the `PROJ` library is not needed.

The PROJ software is freely available. It can be downloaded from its home page at <http://www.remotesensing.org/proj/>.

2.2 The Makefile

The attempt has been made to make the installation of this package as simple and as versatile as possible. The user may install as many of the components he or she wishes simply by editing one script. However, some knowledge of the destination system is required to set some of the needed parameters for complete compilation.

The first step is to edit the **Makefile** found in the distribution's root directory. This is where you set the destination for the compiled programs, the compiler and any compiler flags necessary/desired.

By default, the programs will be installed in the `/usr/local` directory tree. This can be changed by editing the **PREFIX** line in the Makefile, but note that this is not the directory in which the executables will be placed, but one level above that. (Executables will be placed in **BINDIR**.) If the package is to be made available to the whole system, make sure that **PREFIX** is accessible to everyone. This may require that the user have administrator or superuser privileges on the system, depending on the permissions on **PREFIX**. Also make sure that **BINDIR** is included in your **PATH** environment variable so that all the executables can be found when called.

If the PROJ library is already installed, make sure that **PROJPREFIX** is pointing to the base directory for the package, e.g. if the PROJ library is installed in `/usr/local/lib`, use **PROJPREFIX** = `/usr/local`. If PROJ is not already installed, see 2.3.

The compiler to be used is set with **CC**. The package has been compiled with a newer version of **gcc** (i.e. **gcc-2.96**). By default the optimization flag is used (**CCFLAGS** = `-O`), but it can be easily removed if it causes any problems.

Note: Ensure that **X11INCLUDE** and **X11LIB** point to the correct directories. Some distributions put the X11 libraries in different places.

2.3 The PROJ library

Before any of the programs can be compiled, the PROJ library must first be installed on the system. If not already installed, the user must do so. First make sure that variables in the makefile are set appropriately (2.2). Set **PROJPREFIX** to the top directory where you want the library and include files to go (e.g. use `/usr/local` to put the libraries in `/usr/local/lib`). Make sure you have write permission for that directory. Run the command `"make proj"` to compile the libraries and then `"make install_proj"` to install them to the selected directory. Now the rest of the package may be built.

2.4 Compilation

Compilation is done using the **make** command. If the PROJ library is not already installed on the system, see 2.3. To compile the rest of package, just type **make all**. The program will automatically compile the programs in the package, changing directories where necessary. The individual packages may be compiled separately by naming them with **make**, as in the following:

make tri to compile the **triangle** programs; and **make gen** to compile the interpolation and conversion programs.

Individual programs (not including those in the **proj** and **triangle** packages) may be compiled separately, if need be, by typing the name of the program after **make**. For example, **make nei_211** will compile only the **nei_211** program.

2.5 Installation

Installation follows the same pattern as compilation. The programs will be installed in the **BINDIR** directory. To install all the programs, type

make install.

The individual packages may be installed by replacing **install** with either **install_tri** or **install_gen**. The individual programs (not including those in the **proj** and **triangle** packages) may be installed by replacing **install** with **install_<name>**, where **<name>** is the name of the program.

3 Usage

3.1 Before running

Before running the script, the user must edit the script file to set the filenames and the values for the parameters of the mesh generation. The lines to edit are as follows:

3.1.1 Filenames

#Input and output filenames:

```
{bndnod}='eirebnd.nod';    # Input boundary node (Trigrid format) file
{depdat}='eiredep.dat';    # Input depth data file name
{depnod}='deps1';          # Desired root name of depth mesh file
{newfile}='eire_new';      # Desired root name of output neighbour file
```

where `bndnod` is the filename for the boundary node file (Section 5.8), `depdat` is the filename of the depth data file (its format is longitude, latitude, depth, with depth positive downwards), `depnod` is the root of the filename of the output depth mesh and `newfile` is the root of the filename for the finite element mesh neighbour file.

3.1.2 Parameters

Values for the parameters used to generate and refine the mesh.

```
{steiner}="no";            # Allow Steiner points to be added on the
                           # boundary? Default is "no".
                           # If "yes", qangle is used, otherwise
                           # qangle is not used.

{qangle}=0.0;              # Minimum angle between element sides

{dt}=360.0;                # Timestep for wave speed constraint
                           # CFL criterion

{mindp}=20.0;              # Minimum depth for wave speed constraint
                           # When shallower than mindp, the wave
                           # speed constraint is calculated as if
```

```

# it were at mindp.

${dhparm}=1.5;      # Maximum value for slope parameter
                    # i.e. the ratio of the maximum depth to
                    # the minimum depth on an elements nodes

${dhmindp}=20.0;    # Minimum depth for slope parameter
                    # i.e. depths less than this depth will
                    # be treated as being at this depth
                    # when calculating the slope

${dx}=50000;        # Maximum length of element side, used in
                    # setting the initial regular grid

${mindxdp}=250;     # Depth below which the ${mindx} parameter
                    # is used to set the area limit

${mindx}=2000;      # Minimum length of an element side when
                    # the depth of one of the nodes of the
                    # element is greater than ${mindxdp}

${wavefact}=0.1;    # Parameter to set the relative weights of
                    # the slope and wave speed parameters
                    # i.e. The maximum area of the element
                    # is max( slope, wavefact * wave speed )

${iterations}=3;    # Number of times to perform the mesh
                    # refinement

```

The first parameter (**steiner**) is a flag to set the parameter that tells **triangle** (4.11) whether or not to allow adding points (known as Steiner points) on boundaries in order to satisfy constraints imposed on the mesh. The default is to **not** use Steiner points, i.e. no points will be added on boundaries. If Steiner points are allowed then the **qangle** parameter (see below) is also passed on to **triangle**, otherwise it is ignored.

qangle is a parameter for **triangle** (4.11) that sets the minimum angle between sides for the elements in the mesh. According to **triangle**'s web pages, the triangulation algorithm is theoretically guaranteed to terminate for angles less than 20.7 degrees, but usually doesn't terminate for angles above 33.9 degrees. **qangle** is only used when Steiner points are allowed (see above).

The following eight parameters are used by **set_triangle_area** (4.9) to calculate limits on element size which is then used by **triangle** (4.11) to refine the mesh. These parameters are described in section 4.9.

The final parameter is the number of times to perform the refinement process. In our experience a value beyond 3 or 4 showed little difference from the preceding mesh, but you will probably want to do a couple of iterations to get the most effect from the parameters supplied to **set_triangle_area** (4.9).

See Appendix B for example meshes produced by **make_triangle_mesh** based on the above parameters.

3.2 Running the script

To actually run the script, the user only has to enter the command below:

```
> ./make_triangle_mesh
```

The process to create a finite element mesh as performed by **make_triangle_mesh** is as follows:

1. The boundary node file and depth data file are processed so that:
 - (a) a regularly-spaced grid of nodes is placed over the area of the boundary nodes, positions are converted to [x y] coordinates and the boundary and grid nodes are saved in a **trigrd** .nod format file (5.8); and
 - (b) the depths are culled so that they are no closer to each other than the smallest distance between adjacent boundary nodes, the depth positions are converted to [x y] coordinates and the data is saved in a **trigrd** .nod format file (5.8). (**mesh_init** (4.1))
2. The depth data .nod file is converted to a .poly file (5.9) for input into **triangle**. (**nodtopoly** (4.4))
3. The depth data is triangulated, creating the depth mesh and producing a .node file (5.6), a .ele file (5.3) and a new .poly file (5.9). (**triangle** (4.11))
4. The depth mesh is converted to neighbour file format (5.5). (**trngl2nei** (4.12))
5. The boundary .nod file is converted to a .poly file (5.9) for input into **triangle**. (**nodtopoly** (4.4))
6. The .poly regular grid is triangulated by **triangle** (4.11), creating the first version of the mesh. Output files are a **triangle** style .node file (5.6), a .ele file (5.3) and a new .poly file (5.9).
7. For testing purposes, the first mesh is converted to a neighbour file (5.5). (**trngl2nei** (4.12))

Steps 8 - 12 refine the mesh and are repeated a number of times (3 times in the example script, as set in the last line of the parameters section (3.1.2)).

8. The elements (triangles) are reshaped to be more equilateral by moving each of the non-boundary nodes to a new position that is the average position of all the nodes it is connected to. (**reshape_node** (4.8))
9. The depths at the new node positions are interpolated from the original data. (**redep_triangle** (4.6))
10. Limits on the size of each mesh element are calculated based on any of three constraints (bottom slope, shallow water wave speed and the resolution of the nodes on the boundary) and parameters provided by the user. (**set_triangle_area** (4.9))
11. The mesh is re-triangulated, using the element size limits calculated above to refine it. (**triangle** (4.11))
12. For testing purposes, the new mesh is converted to a neighbour file (5.5). (**trngl2nei** (4.12))
13. **reshape_node** (4.8) is run one more time to improve the triangles created during the last triangulation.
14. **triangle** (4.11) is run a final time to ensure a proper mesh.
15. **redep_triangle** (4.6) is run to get the correct depths for each node.
16. The final mesh is converted to a neighbour file (5.5). (**trngl2nei** (4.12))
17. The final mesh file is converted to lat-long coordinates based upon the projection used in **mesh_init**. (**nei_211** (4.2))
18. Finally the depth mesh is converted to lat-long coordinates with the same projection. (**nei_211** (4.2))

An example **make_triangle_mesh** script is given in Appendix A.

After the script is finished, the user may wish to use a mesh editor to perform final checks and adjustments on the mesh. It is possible that stiff elements (elements in which all of the nodes are boundary nodes) and nodes with a large number of neighbours were created. If the mesh is changed, the user can use the **redepc** (4.5) program, included in the package but not used in the perl script, to interpolate the depths from the depth mesh to the edited mesh.

4 Component Programs

These programs are all run automatically according to user specified parameters and files (Section 3.1). The required arguments for the programs are generated in the perl script. The instructions here describe how to run each program as a stand-alone function for information purposes. The user should not need to be concerned with these directions when using the perl script.

4.1 mesh_init

This program creates the initial grid and prepares the depth data to be triangulated.

USAGE:

```
> mesh_init <boundary_file> <depths_file> <dx> <grid_node> <depth_node>
```

This is the first program that is executed by the script and performs two functions. First it creates a regular grid of nodes over the region bounded by the minimum and maximum latitudes and longitudes found in the input, Trigrig format <boundary_file> (5.8). The grid is saved in the Trigrig-style node file <grid_node> (5.8). The spacing of the nodes is given by the <dx> parameter (in metres).

Secondly, the program reads in the depth data, drops data points that are greater than 1/2 times <dx> metres outside a box defined by the minimum and maximum values of the positions found in <boundary_file>, culls the data points so that they are no closer than the minimum distance between adjacent nodes in <boundary_file> and outputs the culled depths in the Trigrig-style node file <depth_node> (5.8).

4.2 nei_2ll

This program converts a neighbour format data file (5.5) in [x y] format to [longitude latitude] format.

USAGE:

```
> nei_2ll <input_file> <output_file>
```

This program uses the PROJ (C.2) cartographic projection library.

NOTE: A parameter file is also needed to perform the conversion. (The parameter file is created automatically in the perl script. It is produced by the program `nei_2xy` (4.3).) This file is necessary to ensure the proper conversion to longitude - latitude coordinates at the end of the script. The file is assumed to have the name <input_file>.proj, where the <input_file> includes the .nei extension, e.g. `fdy_new.4.nei.proj`.

An example is:

```
proj=lcc
lat_1=4.035169e+01
lat_2=4.409710e+01
lon_0=-6.763490e+01
lat_0=4.222439e+01
```

The number of lines in the parameter file depends upon the projection used. In the example (and in the perl script) the projection is Lambert Conic Conformal (**lcc**) and requires four parameters. The [**lon_0 lat_0**] point is the central point around which the conversion is performed and the **lat_1** and **lat_2** are the latitudes which are true scale latitudes for the Lambert Conic Conformal projection.

OUTPUT:

The output is the same neighbour file (5.5) as the input except that the horizontal coordinates are in [**longitude latitude**].

4.3 **nei_2xy**

This program converts a neighbour format data file (5.5) in [**longitude latitude**] format to [**x y**] format.

USAGE:

```
> nei_2xy <input_file> <output_file>
```

NOTE: **mesh_init** includes the function of converting [**longitude latitude**] to [**x y**] and thus **nei_2xy** is not used in the script. However, the program is included here for completion.

This program uses the **PROJ** (C.2) cartographic projection library.

Currently this program assumes a Lambert Conic Conformal projection for the conversion (shown by the **proj=lcc** parameter in the example for **nei_211** (4.2)). The projection is made around a centre point that is the average longitude and latitude calculated from the minimum and maximum values read from the neighbour file. Two true scale latitudes are used in this projection and they are calculated to be 1/4 the range of latitudes.

OUTPUT:

The output is the same neighbour file (5.5) as the input except that the horizontal coordinates are in [**x y**]. Also produced is a parameter file that holds the projection parameters used in the conversion to be used by the **nei_211** program (4.2). The parameter file is given the name **<input_file>.proj**.

4.4 nodtopoly

This program is essentially a preprocessor for the **triangle** mesh generation program. It converts from a **trigrid** .nod format file (5.8) to a .poly format file (5.9).

USAGE:

```
> nodtopoly <nod_filename> <poly_filename>
```

OUTPUT:

A poly format file (5.9) that can be read by the **triangle** and **showme** programs.

4.5 redepc

This program interpolates the depths from one mesh to another. This program is not used in the perl script but is provided so that the user can re-interpolate the depths if the output mesh is edited after running the script. This program can operate on .nei (5.5) and .nod/.ele/.bat (5.7/5.4/5.2) files.

USAGE:

```
> redepc <reference_mesh> <new_mesh> <output_mesh>
```

where **<reference_mesh>** is the mesh to interpolate from, **<new_mesh>** is the mesh to interpolate to and **<output_mesh>** is the mesh with the interpolated depths. Each of the meshes on the command line can be either a .nei (5.5) file or a .nod/.ele/.bat (5.7/5.4/5.2) set of files. If using the .nod/.ele/.bat set of files, all three must be named (e.g. "mesh1.nod mesh1.ele mesh1.bat"). The file formats may be mixed, i.e. the **<reference_mesh>** may be .nod/.ele/.bat files while **<new_mesh>** and **<output_mesh>** may be .nei files.

OUTPUT:

Either a .nei (5.5) file or a .nod/.ele/.bat (5.7/5.4/5.2) set of files, depending on the name(s) entered for **<output_mesh>** on the command line.

4.6 redep_triangle

This program calculates the depths for a mesh outputted from **triangle**. The new depths are interpolated from another mesh, either a neighbour file (5.5) or a NML-style .nod/.ele/.bat (5.7/5.4/5.2) set of files.

USAGE:

```
> redep_triangle
```

or optionally

```
> redep_triangle < <parameter_file>
```

where **<parameter_file>** contains the answers to questions asked by the program.

The first question asked is the format of the original mesh files. Currently the choices are:

- 1) a neighbour file (5.5); or
- 2) a .nod/.ele/.bat set of files (5.7/5.4/5.2).

The user is then asked for the input filename(s). Finally the program asks for the root filename of the **triangle** output files (.node/.ele) (5.6.5.3). The root does not include the extension but does include the version or iteration number given by **triangle**. An example parameter file is:

```
1                # Original mesh file is in neighbour format.
fdy_xy.nei       # Name of the original mesh file.
try3.1           # Root name of the TRIANGLE output mesh files
                  #   to which the depths are to be interpolated.
```

OUTPUT:

Two files are produced by this program:

- 1) a .bat file (5.2); and
- 2) a new **triangle**-style .node file (5.6).

The original .node file is retained by moving it to <root>.node.old. A new .node is created because **triangle** keeps depth info in the .node file.

4.7 regular_grid

Given a .nod (5.8) file containing a boundary (or boundaries), this program will generate a regular grid of points over the domain of the boundary. The spacing of the grid is based on an input parameter.

USAGE:

```
> regular_grid <input_file> <output_file> <max_side_length>
```

where the <input_file> contains the boundary points in .nod format (5.8) and the <max_side_length> is the parameter that sets the spacing between points.

OUTPUT:

The output file is a **trigrid** format .nod file (5.8) containing both the boundary points and the regular grid of points.

4.8 reshape_node

This program attempts to improve the equilaterality of the mesh elements by repositioning each node at the average position of all of its neighbours.

USAGE:

```
> reshape_node <root>
```

where *<root>* is the root filename of the **triangle** .node (5.6) and .ele (5.3) output files.

OUTPUT:

The output file of this program is a new version of the .node file. The original .node file is retained by moving it to *<root>.node.b4.reshape*.

4.9 set_triangle_area

This program is intended for use with the **triangle** mesh generation program. It's purpose is to calculate maximum area constraints for the mesh elements based on parameters provided by the user.

USAGE:

```
> set_triangle_area
```

or:

```
> set_triangle_area < <parameter_filename>
```

If *<parameter_filename>* is not used, the user is asked to enter the root name (the name before the .node/.ele extension) of the input mesh files. Following that, the user is asked whether or not to use each of three constraints: boundary resolution, wave speed and the slope over depth. For each constraint, the user is asked to input parameters for each calculation.

An example *<parameter_filename>* follows:

```
try3          # Root filename of the TRIANGLE output file
y             # Use slope parameter constraint?
y             # Use wave speed constraint?
360.000000    # Timestep for wave speed constraint
20.000000     # Minimum depth for wave speed constraint
y             # Use boundary resolution constraint?
1.500000      # Maximum value for slope parameter
20.000000     # Minimum depth for slope parameter
250.000000    # Depth for ${mindx}
2000.000000   # Minimum length of side when z > ${mindxdp}
0.100000      # Factor for wave speed calculation
              #   to compare to slope calculation
```

OUTPUT:

The output file is *<root>.area* (where *<root>* is the answer to the first question or the first line of the *<parameter_filename>*) and contains a maximum area limit for each element of the mesh.

4.10 showme

This program is part of the **triangle** package and can be used to view the mesh output from **triangle**.

USAGE:

> **showme** *<mesh_filename>*

where *<mesh_filename>* can be any of the output .nod (5.6), .ele (5.3) or .poly (5.9) files or just the root filename with no extension.

4.11 triangle

triangle is the triangulation program used by the script to create the finite element meshes.

USAGE:

> **triangle** *<parameters>* *<grid_filename>*

where *<grid_filename>* is the input file of points to be triangulated. It could be an existing finite element mesh (e.g. for refinement) or a set of points with which a mesh is to be created, as in creating the regular grid in the script.

The parameters used in the script are:

- Y** used to prevent the addition of Steiner points on the boundaries (See 3.1.2);
- a** apply maximum triangle area constraints to the mesh;
- p** read and output a .poly (5.9) file;
- q** add points to the mesh to ensure that no angles smaller than 20 degrees (default) occur (An alternative minimum angle may be supplied after the -q. Should not be used in conjunction with the -Y parameter. (see 3.1.2).); and
- r** refine a previously generated mesh.

OUTPUT:

A .node file (5.6), an .ele file (5.3) and a .poly file (5.9) if the -p parameter is used. The output filenames are based on *<grid_filename>*. If not refining a mesh, the filenames are *<root>.1.<ext>* where *<root>* is *<grid_filename>* without the extension and *<ext>* is node, ele or poly. If refining a mesh (using the -r parameter), the output filenames are the same as above except that the counter or revision number (the .1. above) is incremented, e.g. *<root>.2.<ext>* after the first refinement.

For more information on this program, visit its homepage:

<http://www.cs.cmu.edu/~quake/triangle.html>

4.12 trngl2nei

This program converts the output files from **triangle** (.node, .ele and .poly (5.6, 5.3 and 5.9)) to a neighbour format (5.5) data file.

USAGE:

> trngl2nei <root>

where <root> is the root filename of the **triangle** output files.

OUTPUT:

The neighbour file <root>.nei (5.5).

5 File formats

There are a number of data file formats that are used by the various programs reported here.

5.1 .area files (triangle)

First line: < # elements >

Following lines: < element # > < area limit >

An `.area` file associates with each element a maximum area that is used for mesh refinement. An `.area` file is produced by the `set_triangle_area` program (4.9) with areas calculated based upon parameters given by the user. The file is then used in a subsequent `triangle` (4.11) run to refine the mesh based on physical parameters.

5.2 .bat files (triangle, NML)

All lines: < node # > < depth >

.bat files contains the depth data for each node. These files are produced by the `reddep_triangle` program (4.6). They are not used by any other program in the `make_triangle_mesh` script, but provided for the sake of completeness. .bat files may be used with the `redepc` program (4.5).

5.3 .ele files (triangle)

NOTE: There is a NML version of the .ele file that has a different format. See 5.4.

First line: <# of elements> <# points / element (n)>

<# of attributes (m)>

Following lines: <element #> <pt 1> <pt 2> ... <pt n> <att 1> ... <att m>

where n is the number of points in an element and m is the number of attributes for each element.

.ele files define the elements of the mesh. They are produced by **triangle** (4.11) and used by many programs used here. The number of points per element must be 3. Each point (**pt 1**, etc.) is an index to the corresponding node in the .node file (5.6) and are listed in counterclockwise order around each element. Attributes can be physical quantities (e.g. mass) associated with the elements. During refinement, there is no direct mapping of attributes from input to output elements. Values are interpolated, which may lead to diffusion of the attributes.

5.4 .ele files (NML)

NOTE: There is a triangle version of the .ele file that has a different format. See 5.3.

Each line: <element #> <pt 1> <pt 2> <pt 3>

The NML version of the .ele file has no header and contains one line for each element of the mesh. Points correspond to the node number in the .nod file (5.7) and are listed in counterclockwise order.

5.5 .nei files (Trigrid)

First line: <# of nodes>

Second line: <maximum # of neighbours>

Third line: <maximum x> <maximum y> <minimum x> <minimum y>

Following lines: <node #> <x> <y> <boundary flag> <z> <nei 1> <nei 2>
... <nei n>

This is the format for the final mesh output, and it contains the information found in the .nod (5.7), .ele (5.4) and .bat (5.2) files. The final output of the mesh generation script is in this format. The boundary flag denotes those nodes which are on the external boundary or on islands in the mesh (see 5.6). The list of neighbours lists the nodes that are connected to the current node, sorted in counterclockwise order.

5.6 .node files (triangle)

First line: <# of nodes> <# dimensions> <# of attributes (n)>
<# boundary flags>

Following lines: <node #> <x> <y> <att 1> ... <att n> <boundary flag>

This is the node data file format used by **triangle** (4.11). The number of dimensions must be 2. Attributes are generally floating-point values of physical quantities (such as depth) associated with the nodes. The number of boundary flags is usually 2, i.e. the flag is 0 or 1. 0 denotes an interior node while 1 denotes a boundary node, either the external boundary or an island.

5.7 .nod files (NML)

NOTE: There is a Trigrid version of the .nod file that has a different format. See 5.8.

Each line: <node #> <x> <y>

This is the NML node data file format. There is no header and it contains [x y] or [longitude latitude] pairs for each of the nodes in the mesh.

5.8 .nod files (Trigrid)

NOTE: There is a NML version of the .nod file that has a different format. See 5.7.

```
First line: <# of nodes>
Second line: <# of boundaries (N)>
Third line: <# of nodes for boundary 1 (n1)>
Following n1 lines: <x> <y> <z>
Next line: <# of nodes for boundary 2 (n2)>
Following n2 lines: <x> <y> <z>
...
Next line: <# of nodes for boundary N (nN)
Following nN lines: <x> <y> <z>
Next line: <# of interior nodes>
Following lines: <x> <y> <z>
```

The Trigrid version of the node data file. The two files produced by `mesh_init` (4.1) are in this format. They are converted to .poly (5.9) format for input into `triangle` (4.11) by the `nodtopoly` program (4.4).

5.9 .poly files (triangle)

```
First line: <# of nodes (N)> <# dimensions> <# attributes (n)>
                                         <# boundary flag>
Following N lines: <node #> <x> <y> <att 1> ... <att n> <boundary flag>
Next line: <# segments (M)> <boundary flags>
Following M lines: <segment #> <node #> <node #> <boundary flag>
Next line: <# of holes (K)>
Following K lines: <hole #> <x> <y>
```

A .poly file is used as the input for the first triangulation, as outputted from `nodtopoly` (4.4). It contains the node data and also data on the boundaries (segments) and islands (holes). Segments are defined by their endpoints and their presence is enforced during triangulation. Holes are defined by specifying a point in the hole. After triangulation, `triangle` (4.11) removes elements, starting at the hole point and spreading out until a boundary segment is met. Thus islands are defined with a hole point

surrounded by boundary segments. With the `-p` flag set, `triangle` (4.11) outputs a slightly different `.poly` file. Since the `.node` (5.6) file contains the node data, the `poly` file sets the number of nodes (N) to zero in the first line and thus outputs only the segment and hole data.

A The MAKE_TRIANGLE_MESH Script

```
#!/usr/bin/perl

if ($#ARGV >= 1 ) {
    print "USAGE: make_triangle_mesh \n";
    die("\n");
}

#Input and output filenames:

${bndnod}='eirebnd.nod';    # Input boundary node (Trigrid format) file

${depdat}='eiredep.dat';    # Input depth data file name

${depnod}='deps1';         # Desired root name of depth mesh file

${newfile}='eire_new';      # Desired root name of output neighbour file


# Values for the parameters used to generate and refine the mesh.

${steiner}="no";           # Allow Steiner points to be added on the
                           # boundary? Default is "no".
                           # If "yes", qangle is used, otherwise
                           # qangle is not used.

${qangle}=0.0;             # Minimum angle between element sides

${dt}=360.0;               # Timestep for wave speed constraint
                           # CFL criterion

${mindp}=20.0;             # Minimum depth for wave speed constraint
                           # When shallower than mindp, the wave
                           # speedconstraint is calculated as if
                           # it were at mindp.

${dhparm}=1.5;             # Maximum value for slope parameter
                           # i.e. the ratio of the maximum depth to
                           # the minimum depth on an elements nodes

${dhmindp}=20.0;           # Minimum depth for slope parameter
```

```

# i.e. depths less than this depth will
# be treated as being at this depth
# when calculating the slope

${dx}=50000;          # Maximum length of element side, used in
# setting the initial regular grid

${mindxdp}=250;       # Depth below which the ${mindx} parameter
# is used to set the area limit

${mindx}=2000;        # Minimum length of an element side when
# the depth of one of the nodes of the
# element is greater than ${mindxdp}

${wavefact}=0.1;      # Parameter to set the relative weights of
# the slope and wave speed parameters
# i.e. The maximum area of the element
# is
# max( slope, wavefact * wave speed ).

${iterations}=3;      # Number of times to perform the mesh
# refinement

#####
#####
#                               End User Input Area                               #
#####
#####

if ( ${steiner} eq "Y" || ${steiner} eq "y" || ${steiner} eq "yes" ||
    ${steiner} eq "YES" || ${steiner} eq "Yes" ) {
    ${triparm}="rpq${qangle}"
} else {
    ${triparm}="rpY"
}

# Create the initial, regular grid and cull the depth file.
printf( "\n" );
printf("*** ==> Running mesh_init %s %s %s %s.rtn %s.nod\n", ${bndnod},
    ${depdat}, ${dx}, ${newfile}, ${depnod});
system(
"mesh_init ${bndnod} ${depdat} ${dx} ${newfile}.rtn ${depnod}.nod" )

```

```

    && die "mesh_init failed.\n";

# Convert the depth node file to poly format to prepare for triangle.
printf("\n*** ==> Running nodtopoly %s.nod %s.poly\n", ${depnod},
    ${depnod});
system("nodtopoly ${depnod}.nod ${depnod}.poly") &&
    die "nodtopoly ${depnod}.nod ${depnod}.poly failed.\n";

# Triangulate the depth mesh.
printf( "\n*** ==> Running triangle -pc %s.poly\n", ${depnod} );
system("triangle -pc ${depnod}.poly") &&
    die "triangle ${depnod}.node failed.\n";

# Convert the depth mesh to neighbour format.
printf( "\n*** ==> Running trngl2nei %s.1\n", ${depnod} );
system("trngl2nei ${depnod}.1") &&
    die "trngl2nei ${depnod}.1 failed.\n";

# Prepare the regular grid for triangulation.
printf( "\n*** ==> Running nodtopoly %s.rtnd %s.poly\n", ${newfile},
    ${newfile} );
system("nodtopoly ${newfile}.rtnd ${newfile}.poly") &&
    die "nodtopoly ${newfile}.rtnd ${newfile}.poly failed.\n";

# Triangulate the regular grid
printf( "\n*** ==> Running triangle -p %s.poly\n", ${newfile} );
system("triangle -p ${newfile}.poly") &&
    die "triangle -p ${newfile}.poly failed.\n";

# Convert the regular mesh to neighbour format.
printf( "\n*** ==> Running trngl2nei %s.1\n", ${newfile} );
system("trngl2nei ${newfile}.1") &&
    die "trngl2nei ${newfile}.1 failed.\n";

for ( ${nit} = 1; ${nit} < ${iterations}; ${nit}++ ) {

# Reshape the elements of the mesh (makes better triangles).
printf( "\n*** ==> Running reshape_node %s.%d\n", ${newfile}, ${nit});
system("reshape_node ${newfile}.${nit}") &&
    die "reshape_node ${newfile}.${nit} failed.\n";

open( SETA, '>in.redep_triangle' );

```

```

printf( SETA "1\n%s.1.nei\n%s.%d", ${depnod}, ${newfile}, ${nit} );
close( SETA );

# Interpolate the depths for the new mesh.
printf("\n*** ==> Running redep_triangle < in.redep_triangle (%d)\n",
    ${nit} );
system("redep_triangle < in.redep_triangle") &&
    die "redep_triangle ${nit} failed.\n";

# Use set_triangle_area to refine the elements of the mesh base on
# parameters given above.
open( SETA, '>in.set_triangle_area' );
printf( SETA "%s.%d\n", ${newfile}, ${nit} );
printf( SETA "Y\n" );          # Use slope parameter constraint?
printf( SETA "Y\n" );          # Use wave speed constraint?
printf( SETA "%f\n", ${dt} );  # Timestep for wave speed
                                # constraint
printf( SETA "%f\n", ${mindp} ); # Minimum depth for wave speed
                                # constraint
printf( SETA "Y\n" );          # Use boundary resolution
                                # constraint?
printf( SETA "%f\n", ${dhparm} ); # Maximum value for slope
                                # parameter
printf( SETA "%f\n", ${dhmindp} ); # Minimum depth for slope
                                # parameter
printf( SETA "%f\n", ${mindxdp} ); # Depth for ${mindx}
printf( SETA "%f\n", ${mindx} );  # Minimum length of side when
                                # z > ${mindxdp}
printf( SETA "%f\n", ${wavefact} ); # Factor for wave speed
                                # calculation to compare to

    # slope calculation
close( SETA );

printf(
"\n*** ==> Running set_triangle_area < in.set_triangle_area (%d)\n",
    ${nit});
system("set_triangle_area < in.set_triangle_area") &&
    die "set_triangle_area ${nit} fialed.\n";

# Refine the mesh based on the area limits calculated above.
printf( "\n*** ==> Running triangle -a%s %s.%s\n", ${triparm},
    ${newfile}, ${nit} );

```

```

system("triangle -a${triparm} ${newfile}.${nit}") &&
die "triangle -a${triparm} ${newfile}.${nit} failed.\n";

${nit1}=${nit} + 1;

# Convert the new mesh to neighbour format.
printf( "\n*** ==> Running trngl2nei %s.%s\n", ${newfile}, ${nit1} );
system("trngl2nei ${newfile}.${nit1}") &&
die "trngl2nei ${newfile}.${nit1} failed.\n";
}

${nit1}=${iterations};
${nit2}=${iterations} + 1;

# Perform the final reshape.
printf( "\n*** ==> Running reshape_node %s.%s\n", ${newfile},${nit1} );
system("reshape_node ${newfile}.${nit1}")&&
die "reshape_node ${newfile}.${nit1} failed.\n";

printf( "\n*** ==> Running triangle -%s %s.%s\n", ${triparm},
        ${newfile}, ${nit1} );
system("triangle -${triparm} ${newfile}.${nit1}") &&
die "triangle -${triparm} ${newfile}.${nit1} failed.\n";

# Interpolate the depths for the final mesh.
open( SETA, '>in.redep_triangle' );
printf( SETA "1\n%s.1.nei\n%s.%d", ${depnod}, ${newfile}, ${nit2} );
close( SETA );

printf( "\n*** ==> Running redep_triangle < in.redep_triangle (#%d)\n",
        ${nit2} );
system("redep_triangle < in.redep_triangle") &&
die "redep_triangle ${nit2} failed.\n";

# Convert the final mesh into neighbour format.
printf( "\n*** ==> Running trngl2nei %s.%s\n", ${newfile}, ${nit2} );
system("trngl2nei ${newfile}.${nit2}") &&
die "trngl2nei ${newfile}.${nit2} failed.\n";

# Convert the final mesh back into lat-long coords.
# Coords were converted into x-y in mesh_init.
printf( "\n*** ==> Running nei_21l %s.%s.nei %s_1l.%s.nei\n", ${newfile},

```

```

        ${nit2}, ${newfile}, ${nit2} );
system( "mv ${bndnod}.proj ${newfile}.${nit2}.nei.proj" ) &&
    die "mv ${bndnod}.proj ${newfile}.${nit2}.nei.proj failed.\n";
system("nei_2ll ${newfile}.${nit2}.nei ${newfile}_ll.${nit2}.nei") &&
    die
        "nei_2ll ${newfile}.${nit2}.nei ${newfile}_ll.${nit2}.nei failed.\n";

# Convert the depth mesh into lat-long coords.
printf( "\n*** ==> Running nei_2ll %s.1.nei %s_ll.1.nei\n", ${depnod},
        ${depnod} );
system( "cp ${newfile}.${nit2}.nei.proj ${depnod}.1.nei.proj" ) &&
    die "cp ${newfile}.${nit2}.nei.proj ${depnod}.1.nei.proj failed.\n";
system( "nei_2ll ${depnod}.1.nei ${depnod}_ll.1.nei" ) &&
    die "nei_2ll ${depnod}.1.nei ${depnod}_ll.1.nei failed.\n";

```

B Examples of the script in action

Figure 1 shows the triangulated depth mesh used to refine the mesh. This example was built from an existing mesh and so the depth mesh has regularly spaced points at varying resolutions. However, depth meshes can have regularly spaced points, randomly spaced points or some combination thereof.

In Figures 2-5 the results of the example script are presented, starting with the regular grid and showing the results after each refinement.

Figure 6 shows the final mesh from the example script when Steiner points are allowed on the boundary and the angle constraint is used. (See 3.1.2 for a description of this option for the script.)

The figures in this appendix were created using the `showme` (4.10) program that is included in the `triangle` package.

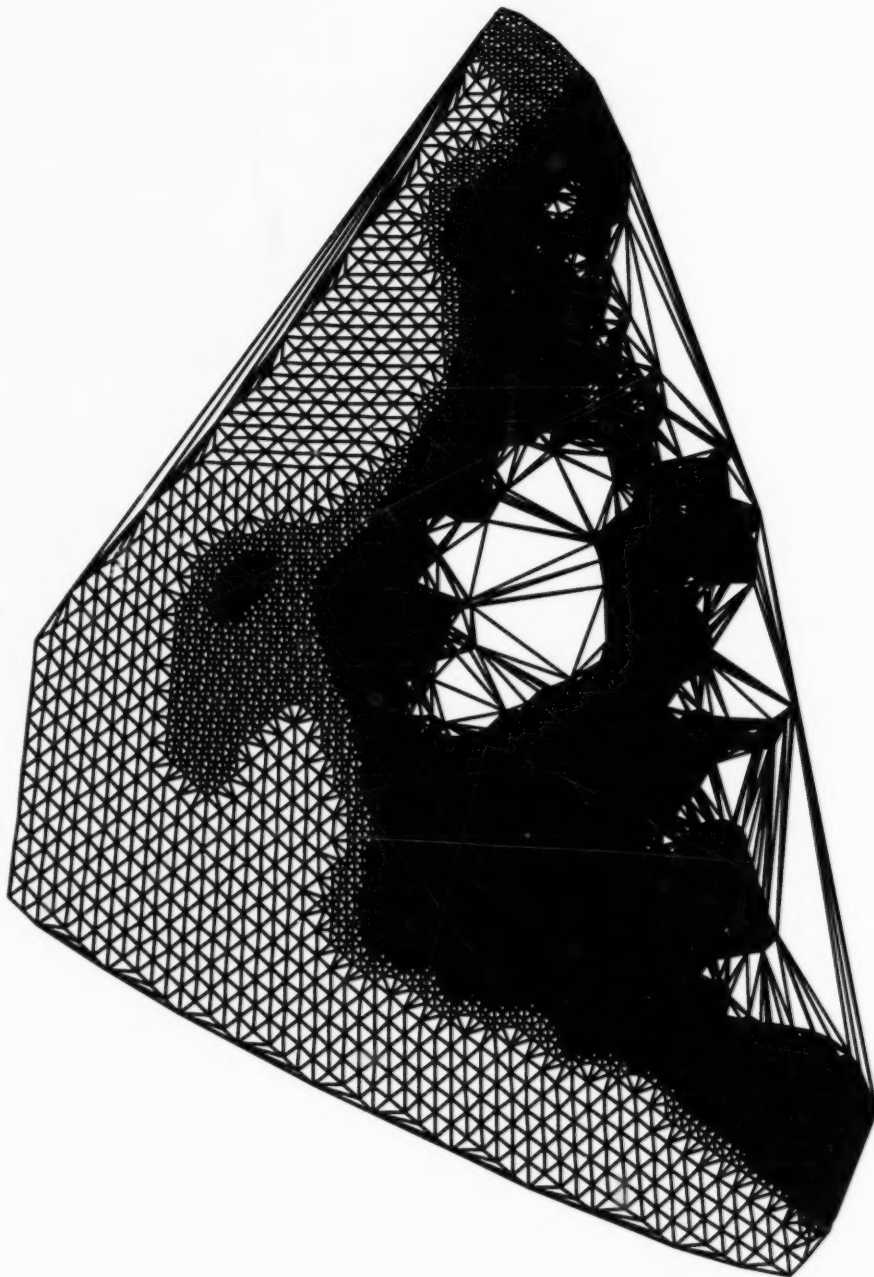


Figure 1: The triangulated depth mesh. There are 29155 nodes and 58280 elements.



Figure 2: The mesh after the initial triangulation. There are 6452 nodes and 7025 elements.

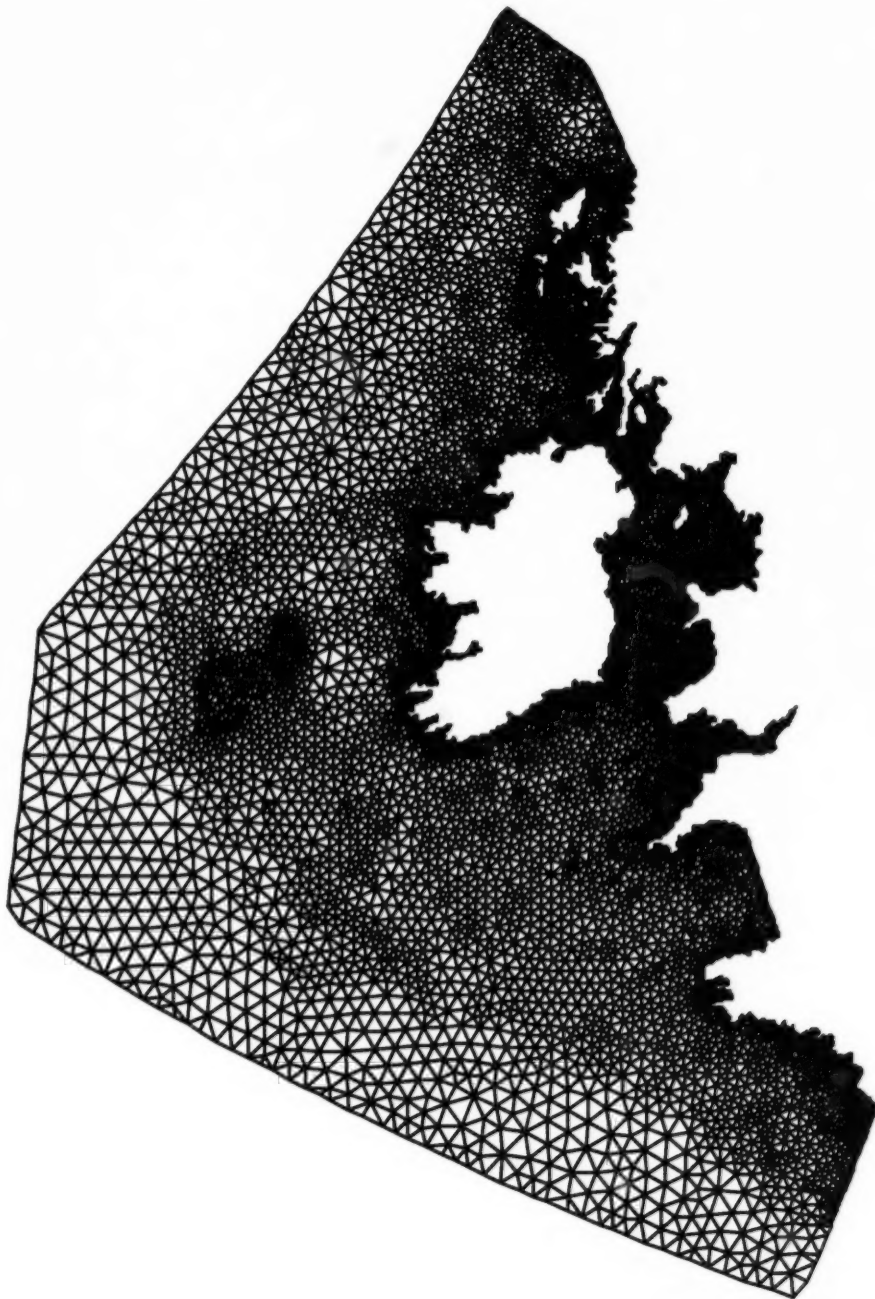


Figure 3: The mesh after the first refinement. There are 14241 nodes and 22603 elements.



Figure 4: The mesh after the second refinement. There are 15337 nodes and 24795 elements.

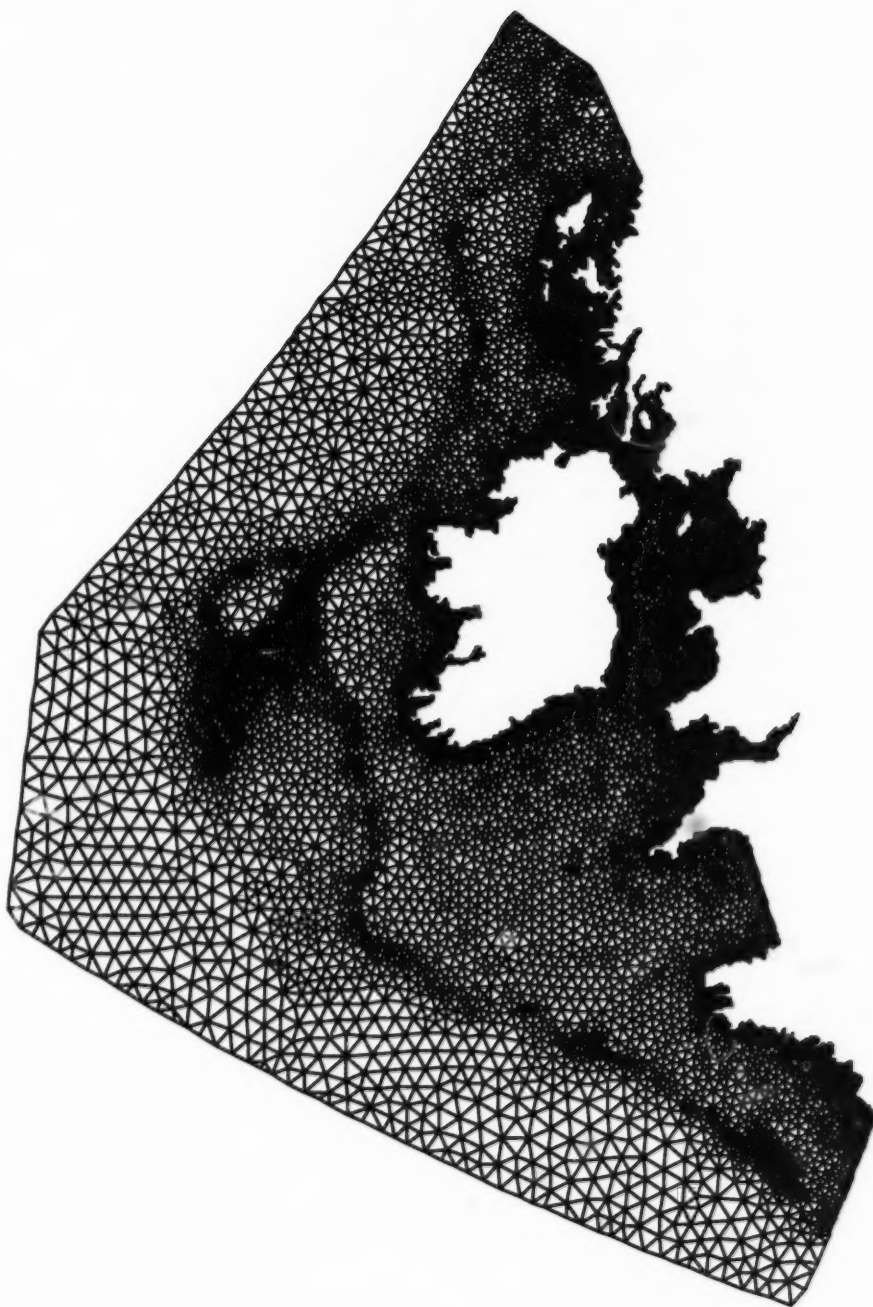


Figure 5: The mesh after the third refinement. There are 15337 nodes and 24795 elements.

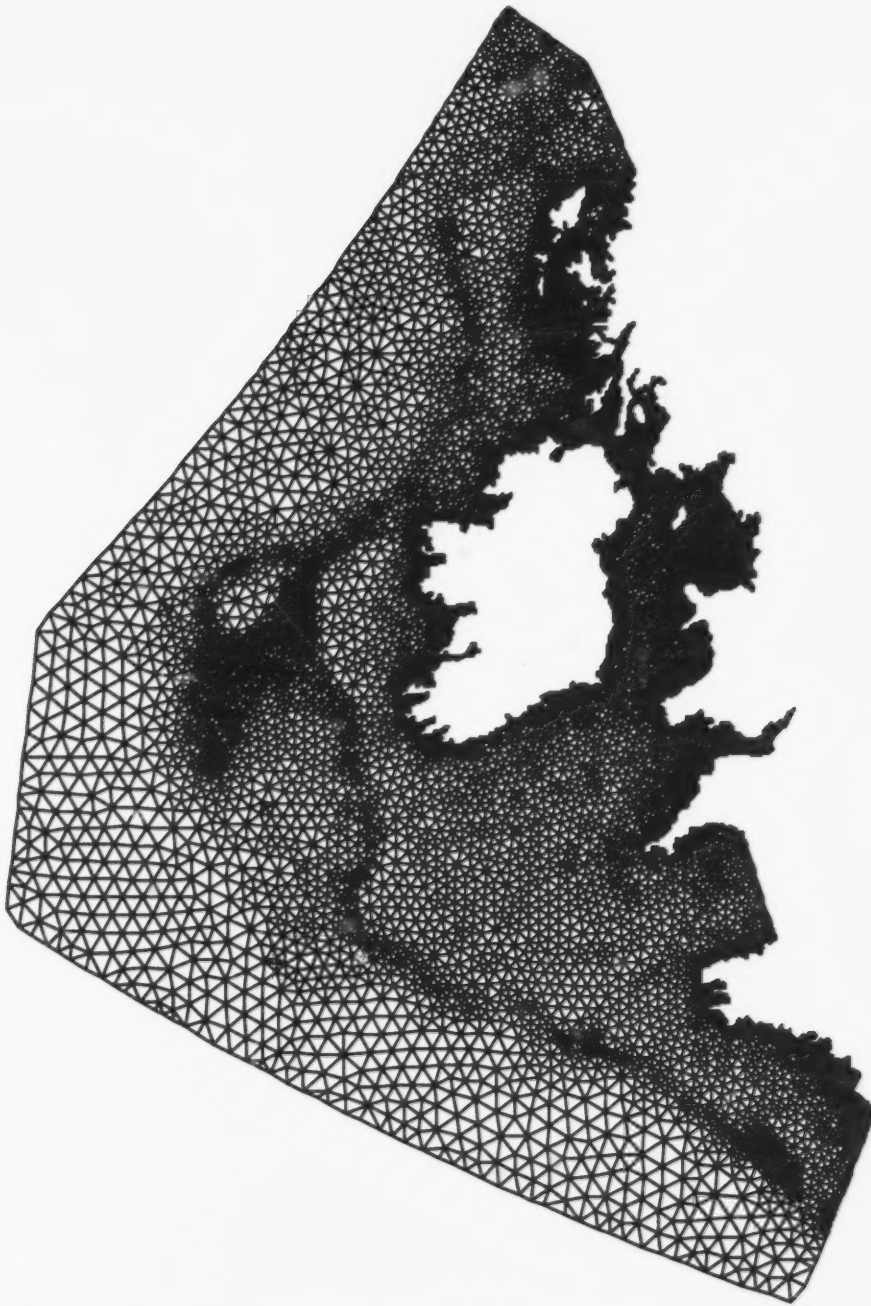


Figure 6: The mesh after the third refinement when allowing the use of Steiner points on the boundary. (See 3.1.2.) There are 16550 nodes and 26484 elements. A **qangle** of 0 degrees (See 3.1.2.) was used as it was the only value for which the first refinement of the mesh succeeded.

C Web Sites

Here are the web sites for the programs and libraries mentioned in this document. These pages provide download links and reference material that has greater depth than this document.

C.1 resolute

The entire mesh generation package described in this document is available through anonymous ftp from `starfish.mar.dfo-mpo.gc.ca` in the `/pub/ocean/dgreenbe` directory. The package's name is `resolute_1.1.tar.gz`.

C.2 PROJ

PROJ is the Cartographic Projections library used to convert positions in latitude and longitude co-ordinates into some user-defined `[x,y]` co-ordinates.

The home page for PROJ is :

<http://www.remotesensing.org/proj/>

C.3 Triangle

Triangle is described as a 2-D quality mesh generator and Delaunay triangulator. Also included in the package is the program `showme`, used to view the meshes created by `triangle`.

The home page for Triangle is :

<http://www-2.cs.cmu.edu/~quake/triangle.html>

It is also available from :

<http://www.netlib.org>

in the `voronoi` directory.

C.4 TriGrid

TriGrid is a system for building triangular grids. It is not used by the mesh generation script described in this document, but some of its data file formats are used. However, it could be useful for editing meshes that have been created.

The home page for TriGrid is :

<http://trigrid.sourceforge.net/>

